Modularized Exception Handling

Martin S. Feather
JPL - Mail Stop 125-233, 4800 Oak Grove Drive, Pasadena CA 91109-8099, USA
Email: feather@jpl.nasa.gov
(This work was performed while at USC/Information Sciences Institute)

Abstract

When programs share data, it would be convenient if those programs could make their own assumptions about that data, without requiring the data to satisfy all the programs' assumptions simultaneously. Even when the same assumption *is* shared by several programs, it would be convenient to allow those programs to treat violations of that assumption differently. One an approach towards achieving this is to give each program its own view of the shared data.

A mechanism implementing this approach is described. It derived each program's view from the shared data so as to satisfy all that program's assumptions, by a user-specified combination of ignoring facts that hold in the shared data and feigning facts that do not hold.

In general, any approach to such modularizing of exception handling must strive to meet several desiderata. These are presented, and the degree to which this particular mechanism meets them is described. Some alternative approaches, offering different degrees of achievement of these desiderata, are briefly discussed.

1 Motivation and Outline

Consider programs sharing a database of personnel information. Suppose that some of those programs make the assumption that only workers who have security clearances are working on classified projects. If this assumption is imposed on the shared database, then the database is precluded from holding data violating it (e.g., uncleared worker `Martin` working on classified project `Stealth`). Conversely, if such violating data is allowed to reside in the shared database, then every program must be able to tolerate its presence. Even if this is acceptable, it might be the case that the programs making the assumption might wish treat violations in different ways (e.g., one program might ignore the fact that `Martin` works on project `Stealth`; another might feign the fact that `Martin` is cleared; yet another might ignore the fact that `Stealth` is classified). As a different example, consider a shared pool of epidemiological data (e.g., heights, weights, ages, etc., of people grouped into various treatment classes). Some programs may assume that every person has a height recorded in the database; of these, one may wish to substitute its own default value when the a person's height value is missing, while another program might wish to ignore that person entirely when performing analysis.

To accommodate multiple programs sharing common data, one possible solution is to give each program its own *view* of the shared data. Each program's view is derived from the shared data in such a way as to satisfy all that program's assumptions, and to treat violations of those assumptions in the manner required by that program. For example, a program assuming that only workers who have security clearances are working on classified projects could be given a view in which every uncleared worker who works on a classified project appear to be cleared. In the remainder, section 2 outlines a prototype mechanism that implements this view-based approach. Section 3 discusses some of its implementation details. Section 4 presents a set of desiderata likely to be shared by any solution to the overall problem, and considers the extent to which the particular mechanism described herein meets them. Section 5 outlines the key capabilities required for the rapid prototyping of this mechanism. Section 6 considers some related work, and finally section 7 offers some speculations based on this experience.

2 MEH - a Prototype Mechanism for Modularized Exception Handling

A mechanism, called **MEH** herein, has been prototyped. This section provides an overview of its operation from the user's point of view.

2.1 Overview of MEH

MEH supports arbitrary assumptions (any assumption that can be expressed as a predicate over the database), but only a limited range of violation handlers — these work by *ignoring* facts within the shared database, and/or by *feigning* facts to hold that do not hold in the shared database. The programmer is required to provide simple declarations of all the assumptions imposed by his/her program, and corresponding violation handlers. From these declarations, **MEH** constructs automatically the code that:

- derives that program's view of the shared database;
- causes the program's updates to be attempted on the shared database — **MEH** admits the possibility that

one program's updates violates another program's assumptions and cannot be handled by the corresponding violation handlers; in such a case, the transaction performing those updates is not allowed to occur, and the data is left unchanged (as would be the case if the program violated one or more of its own assumptions and the handler could not compensate).

2.2 Declaration of Assumptions and Violation Handling

MEH requires all of a program's assumptions to be declared. Each such declaration is local to its program, and comprises a name and definition (an arbitrary predicate over the data as viewed by the program).

EXAMPLE: Suppose that a program needs to assume that: *every worker who works on a classified project must be cleared*: The assumption is declared to **MEH** as follows[1]:

```
assumption CLASSIFIED-REQUIRES-CLEARED(w,p) =
   ( WORKER(w) ∧ PROJECT(p) ∧
     WORKS-ON(w,p) ∧ CLASSIFIED(p) ) ⊃
   CLEARED(w)
```

`CLASSIFIED-REQUIRES-CLEARED` is the name being given to this assumption. Its formal parameters, `w` and `p` together with its body `(WORKER(w) ...)` ⊃ `CLEARED(w)`, define the following predicate universally quantified over those parameters:

```
∀ (w,p) ( WORKER(w) ∧ PROJECT(p) ∧
          WORKS-ON(w,p) ∧ CLASSIFIED(p) ) ⊃
        CLEARED(w)
```

Separately, **MEH** requires the declaration of violation handlers, which will be applied when the shared database's data does not satisfy all of the program's assumptions. They will be used to construct a view in which facts in the shared database are ignored, and/or facts not in the shared database are feigned, in the view of the data seen by the program.

EXAMPLE (continued): suppose the aforementioned program wishes to treat violations of the assumption (instances of uncleared workers working on classified projects) by *ignoring* the works-on relation between uncleared worker and classified project. This is declared to **MEH** as follows:

```
handler CLASSIFIED-REQUIRES-CLEARED(w,p) =
 { ignore WORKS-ON(w,p) }
```

The handler is given the same name as the assumption whose violations it will respond to. Its definition consists of a set (in this example a singleton set) whose elements are of the form `ignore <relation-name>(<tuple of objects>)`, and/or `feign <relation-name>(<tuple of objects>)`. Every binding of objects to parameters that causes the assumption's predicate to evaluate to false (i.e., is an assumption violation) is used to instantiate the variables of the handler's set.

EXAMPLE (alternative): consider a different program making the same assumption as before, but wishing to treat violations by by *feigning* that those (uncleared) workers working on classified projects are cleared. This would be declared to **MEH** as follows:

```
handler CLASSIFIED-REQUIRES-CLEARED(w,p) =
 { feign CLEARED(w) }
```

[1] All the boxed declarations and code fragments in this paper are examples run through **MEH**. The parenthesis-dominated Lisp-like syntax it employs has been re-expressed in a more widely palatable syntax for readability. Relation queries take the form <relation-name>(<arguments>), for example, WORKS-ON(w,p) is a query of whether the 2-tuple of objects referred to by w and p is in the WORKS-ON relation.

3 Implementation of MEH

In order to give each program its own suitably derived view of the shared database, **MEH** generates, for each program, a new definition local to only that program of every relation feigned and/or ignored by one or more of that program's assumptions' repairs.

Each such new local relation definition is expressed in terms of the corresponding relation in the shared database, and two new relations (also local to only that program) holding facts to be ignored/feigned by the program.

For example, if binary relation `WORKS-ON` is feigned and/or ignored in a program's handler, **MEH** creates the following definitions local to that program:

```
relation feign-WORKS-ON

relation ignore-WORKS-ON

relation WORKS-ON(w,p) =
   feign-WORKS-ON(w,p) ∨
   SDB::WORKS-ON(w,p) ∧
   ¬ ignore-WORKS-ON(w,p)
```

The above uses the underlying database system's capability for defining relations in terms of other relations in the database. The general form of this construction is:

relation <name>(<n-tuple of variables>) =
    <predicate over those variables>

For any n-tuple of objects in the database, the relation so defined will hold of that tuple if and only if its defining predicate evaluates to true for that tuple.

Thus `WORKS-ON` is defined to hold of a 2-tuple of objects `w` and `p` if and only if the following predicate holds of those objects:

```
feign-WORKS-ON(w,p) ∨
SDB::WORKS-ON(w,p) ∧
¬ ignore-WORKS-ON(w,p)
```

`SDB::WORKS-ON` is the way of referring to the `WORKS-ON` relation of the shared database (`SDB` is mnemonic for `S`hared `D`ata`B`ase). Note that `ignore-WORKS-ON` and `feign-WORKS-ON`, have been defined local to this particular program; other programs may have their own local versions of these relations holding different sets of tuples.

The appropriate namespace declarations are generated to ensure that all references to relation `WORKS-ON` within the text of this program will now refer to this locally defined version, while the original `WORKS-ON` relation of the shared database being hidden from view, while all relations of the shared database which are not redefined locally are left visible to the program. This provides the program a view of the shared database without requiring *any* modification of the text of the program.

These `feign-` and `ignore-` relations are populated by code which watches for violations of the program's assumptions in the program's view of the shared database, and reacts to the detection of such violations by making assertions

into those two relations as indicated by the violated assumptions' repairs. This code is also generated automatically by the implementation. Any assumption for which there is no corresponding handler is turned into code that watches for violations, but does not make any assertions in response to their occurrence.

Briefly, the operation of this code is such that each time a transaction (update to the shared database) is attempted, the contents of all the `ignore-` and `feign-` relations are re-computed. This is done by:

I. Initializing all the `ignore-` and `feign-` relations to be empty of all tuples (thus restoring each programs' view to be identical to the shared database itself).

II. Having each program's assumption code looks for violations in its view of the shared database. If no violations are found, the transaction has been successfully carried out, yielding a new state of the shared database from which every program has a derived view satisfying all of its assumptions. If violations are found, then the corresponding handlers' assertions to `feign-` and/or `ignore-` relations are gathered. All these assertions are simultaneously added to the initiating transaction, i.e., yield a new state of the shared database in which the assertions of the original transaction *and* all of these additional assertions have been carried out (thus likely changing the programs' views, since those views are defined in terms of `feign-` and `ignore-` relations).

Step 2 is repeated until either the process has terminated successfully, or one of the following unsuccessful termination conditions occurs:

- For some tuple of objects and relation `R`, both `ignore-R` and `feign-R` have been asserted of that same tuple.
- No new assertions of `ignore-R` and `feign-R` have been added (thus repeating step 2 will have no effect whatsoever, i.e., an infinite loop).
- The number of repetitions of step 2 has reached some pre-set limit (a crude approximation of an infinite loop check).

Unsuccessful termination causes the transaction that initiated to be retracted (i.e., the database is restored to the state it was prior to starting this transaction), and the program attempting the transaction is informed of its failure.

It is important to realize that only the code introduced by **MEH** ever sees a database in any of the intermediate states of this iterative process — the original programs will always find themselves operating in views satisfying all of their assumptions.

The overall approach requires these updates to be conducted on the shared database (so that this remains as a medium of communication among the programs). Hence, in defining the local relation, **MEH** adds the appropriate code to cause an update to these locally defined relations to be passed on as an update of the corresponding relation in the shared database.

4 Desiderata

4.1 Desiderata

In the quest of permitting programs sharing data to impose not-necessarily-identical sets of assumptions on that data, and to allow violations of assumptions to be treated differentially by different programs, the following are the critical desiderata:

I. Convenience: modification of those programs to fit the approach should be easy.

II. Efficiency: any additional run-time burden (both time and space) imposed by the approach should be minimized.

III. Applicability: as wide a range of assumptions and treatment of violations as possible should be supported.

IV. Connectedness: the shared data should be retained as a medium of communication between those programs; it should be updatable by any of them.

V. Unhindered: the updates made by one program should not be hindered by the assumptions made by another program.

VI. Determination of divergence: an important aspect of the view-based approach is that each program's view can be different from actual data being shared. Thus two programs might be seeing widely different views. It should be possible to determine the degree of divergence between a program's view and the shared data, or between two programs' views.

Note that if efficiency and connectedness are unimportant for some application, it would suffice to replicate the data, giving each program its own entire copy on which it could apply traditional exception handling mechanisms.

4.2 MEH's desiderata tradeoffs

MEH achieves these desiderata to varying degrees of success, as outlined next:

Convenience: the programmer is required to provide **MEH** with declarations of the assumptions and repairs (tuples to be feigned or ignored) to violations of those assumptions. These declarations are relatively straightforward, and from them **MEH** automatically constructs all the additional code required. In particular, the programmer need not make *any* changes to the program text. **MEH** guarantees that the program will operate in a view in which all of its assumptions are always met.

Efficiency: there are two sources of run-time performance penalty when **MEH**'s code is employed. 1) Every query by a program of a relation local to that program's view (i.e., whose tuples may be ignored and/or feigned in that relation's view) must go through the computation involving the `ignore-` and `feign-` relations associated with that relation. 2) an update to a relation for which there is a program with its own local view of that relation necessitates the recomputation of each such program's entire view. In many circumstances this complete recomputation is unnecessary, and a more incremental computation of the view would suffice — pursuit of this is future work.

Applicability: any assumption that can be stated as a predicate over the database can be dealt with by **MEH**. Each program will be guaranteed a view in which its assumptions always holds. This gives **MEH** wide applicability. Handling of assumption violations is achieved by feigning and ignoring of tuples. While in principle *any* view could

be derived by a suitable combination of ignoring and feigning of tuples, the convenience of doing so rests upon the ease of expressing this as a combination of ignores/feigns responses, and as such may be a practical limitation.

Connectedness: the shared database does remain as a medium of communication between the programs. Because programs' updates are passed on to and carried out in the shared database, and because programs' views are derived from the shared database, they will see immediately the effects of each others' updates (suitably modified through their own views, of course).

Unhindered: **MEH** does *not* guarantee that one program's updates be unhindered by another program's assumptions. If an update is such that not all programs are able to recompute a view consistent with their assumptions, then that update is not allowed to happen, the database is left unchanged, and the program making the update is informed of the failure. An open question regarding **MEH** is to be able to analyze sets of assumption so as to determine whether (and if so under what conditions) updates might be so hindered.

Determination of divergence: Using **MEH** determintation of divergence between different programs' views, or between a program's view and the contents of the shared data base, is straightforward. For example, if the values in the `WORKS-ON` relation might differ between a program's view and the shared data base, then the following query, when evaluated in the program's view, will retrieve all the tuples in the shared data base but *not* in the program's view:

```
(listof (w,p) s.t.
   SDB::WORKS-ON(w,p) ∧ ¬ WORKS-ON(w,p))
```

(Recall that by issuing this query in the program's view, `WORKS-ON(w,p)` refers to the relation as viewed by the program while `SDB::WORKS-ON(w,p)` refers to the relation in the shared database.)

5 Key Capabilities for Implementation

MEH is built on top of ISI's in-house virtual memory relational database AP5 [3], an extension of CommonLisp. However, the key capabilities employed to construct **MEH** are common; they are as follows:

Simple modularization of namespaces — built in terms of the package facility of CommonLisp [9]. This allows placement of relation definitions in the namespaces local to particular programs, thus "shadowing" (hiding) the same-named relation residing in the shared database. Any reasonable modularization facility would offer the necessary capabilities to do this. Lacking such a capability, an alternative approach would be to pre-process the program texts, altering all their uses of the affected relations to refer instead to the newly defined relations, i.e., a readily-automated modification of program texts.

Defining a database relation in terms of other database relations — such a capability (known as providing *derived data*) is commonly supported in database work (e.g., see [5]). **MEH** employs the additional capability to have an update to such a defined relation be passed on as an update to some other relation. If this additional capability were not available, simple pre-processing of program texts would again suffice as an alternative means of achievement.

Constraints (also referred to as *integrity conditions*, *invariants* or *consistency checks*) and accompanying *transaction/rollback* mechanism that guard a database from entering a state violating one or more constraints. Any transaction that causes a violation is prevented from occurring, and the database is left in the state it was prior to starting that transaction.

A *repair mechanism* for constraints that adjusts a constraint-violating transaction to (try to) result in a transaction that meets all the constraints. Event-condition-action capabilities of databases [7] and [4] offer the necessary building blocks from which to construct a mechanism such as needed here.

6 Related Work

6.1 Tolerating Exceptions

Established mechanisms for handling of persistent exceptions (ones that must remain present in the data for some extended period of time) are generally intended for a context in which a program or programs all deal uniformly with violations of the data in a single database.

Borgida did early work on handling persistent violations of assumptions [2]. One of the ideas proposed there is that of *blaming* a violation on one or more of the database facts; this **MEH**'s motivation for violation handling by *ignoring* or *feigning* certain database facts as viewed by programs.

Borgida allows such violations to persist by adjusting the definition of the constraint just enough to tolerate them. He uses assignment of blame to indicate which database fact is the "unusual" one. This indication can then be used in two ways: by the program, provided it has been designed to look for and treat such unusual facts accordingly, and by the database, to adjust the definition of the constraint to tolerate that violation in the data. **MEH**'s ability to ignore facts present in the database or feign facts not present is similar to Borgida's second use of blamed facts, but rather than adjusting the constraint to tolerate the violation, I instead use the constraint to generate the definitions that in turn adjust the program's view of the facts in the database. The key distinction is that **MEH** requires no modification to the program code. Borgida's approach would seem to be better suited to incrementally accommodating exceptional data as it arises at run-time.

Balzer suggested the treatment of persistent violations to assumptions by having the data involved in them automatically marked as *polluted* [1]. The data-accessing code can then look for data marked as polluted, and treat it accordingly. The marking of polluted data is performed automatically, the code to do this derived from a declaration of the assumption. This works for any assumption, but requires the explicit modification of the program code to look for, and react accordingly to, polluted data.

6.2 Multiple level secure databases

In the world of multiple level secure databases there are levels of security ('top-secret', 'secret', etc.) assigned to data ('classification') and to users ('clearance'). In this context, users must never be able to access data of a higher level of classification than their own clearance. Thus users with different levels of clearance sharing the same database see different versions of the data. View-based approaches have been applied, where the data in the shared database is

marked with its classification (in some implementations this marking can be as fine-grained as on a tuple-by-tuple basis), and the user's view derived from this shared database contains only the data classified at or below that user's clearance level. See [6] for an instance of this work; [10] for a discussion of 'polyinstantiation', the term used by this community to refer to the issue of giving different views of the same set of facts to different users, and [8] (section 6.1 in particular), for a recent taxonomy.

There are clear similarities between the desiderata suggested in this paper, and those needed by the security community. The key differences are:

- **MEH** supports a wide range of assumptions and corresponding treatments, whereas the secure database community addresses a restricted range — primarily the hiding ('ignoring' in my terms) of certain information from certain users.
- **MEH**'s ability to answer a user's queries about differences between his/her view and the shared database would, of course, have to be disabled.
- The security community's hiding concerns encompass not only explicit queries, but also 'covert channels' (any usage of the system through which information may flow to a user with less clearance than required). The latter have not been addressed at all in **MEH**. For example, **MEH**'s response time to a query may well vary depending on whether or not information was being ignored, whereas in the secure database world this might be a means by which a user could deduce the presence, and perhaps contents, of information supposed to be hidden from him/her. Another implication of avoiding covert channels is that any transaction that would be valid in a user's view should not be precluded because of a violation involving data and/or assumptions of some higher level of clearance. Deciding when this is the case remains an open question in **MEH**.

7 Speculations

Rapid prototyping of **MEH** (and Balzer's own mechanism [1]) was made relatively straightforward by building on top of a relational database offering derived data, general constraints, and transaction repair mechanisms. This suggests that for the purposes of exploration and experimentation of viewpoint resolution mechanisms, mechanisms provided by advanced databases provide an effective starting point.

Handling of exceptions in data shared among multiple programs raises a number of (somewhat contradictory) desiderata. The mechanism presented, **MEH**, is but one possible solution, with its own set of strengths and weaknesses with respect to these desiderata. Other mechanisms exhibit other tradeoffs, and further exploration of the space of possibilities is desirable. A clear categorization of approaches to exception handling, such as that begun by Etzion [4], would be most helpful in revealing the relationships between this growing body of work.

Acknowledgements This research was conducted at USC/ISI, supported by Advanced Research Projects Agency contract No. F30602-93-C-0240. I particularly thank Don Cohen for AP5 support, Bob Balzer and Dave Wile for discussions on violation handling, and Bashar Nuseibeh for feedback and encouragement. Views and conclusions in this document are those of the author and should not be interpreted as representing the official opinion or policy of JPL, NASA, ARPA, the U.S. Government, or any other person or agency connected with them.

References

[1] R. Balzer. Tolerating Inconsistency. In *Proceedings, 13th International Conference on Software Engineering, Austin*, pages 158–165, Texas, USA, August 1991. IEEE Computer Society Press.

[2] A. Borgida. Language Features for Flexible Handling of Exceptions in Information Systems. *ACM Transactions on Database Systems*, 10(4):565–603, December 1985.

[3] D. Cohen. Compiling complex database transition triggers. In *Proceedings, ACM SIGMOD International Conference on the Management of Data*, pages 225–234, Portland, Oregon, 1989. ACM Press. SIGMOD RECORD Volume 18, Number 2, June 1989.

[4] O. Etzion. A reflective approach for data-driven rules. *International Journal of Intelligent and Cooperative Information Systems*, 2(4):399–424, 1993.

[5] S. Koenig and R. Paige. Control of derived data. In *Proc. VLDB*, pages 306–318, 1981.

[6] T.F. Lunt, D.E. Denning, R.R. Schell, M. Heckman, and W.R. Shockley. The SeaView Security Model. *IEEE TSE*, 16(6):593–607, June 1990.

[7] D. McCarthy and U. Dayal. The architecture of an active data base management system. In *Proceedings of the 1989 SIGMOD conference*, pages 215–224, 1989.

[8] M.S. Olivier and S.H. von Solms. A Taxonomy for Secure Object-Oriented Databases. *IEEE Transactions on Database Systems*, 19(1):3–46, March 1994.

[9] G.L. Steele. *Common Lisp: the language*. Digital Press, 1984.

[10] M.B. Thuraisingham. Mandatory Security in Object-Oriented Database Systems. In *Proceedings of the Conference on Onject-Oriented Programming Systems, Languages and Applications*, pages 203–210. ACM, 1989.